

**MANİSA CELAL BAYAR UNIVERSITY**

**CSE 3213**

**ARTIFICIAL INTELLIGENCE**

**HOMEWORK: Local Search Algorithms**

**SUBJECT: 0-1 Knapsack Problem**

*170315022 – EKİN ŞUATAMAN*

*18 January 2021*

# A – Problem Formulation

## 1) State Representation

For hill climbing search and genetic search i have created two arrays that hold iight and values. Then i asked the user for the number of items and the capacity of the bag. After that i used for loop structure to get the weights and values of all items.

I searched the converting structure on the internet to convert the strings to a list.

Likewise, i used the “listToString” method to convert lists to strings.

```
list_of_weights= []
list_of_values = []

numberOfItems = int(input("How many items will you enter?"))
knapsack_capacity = int(input("What will be knapsack_capacity(Bag capacity)?"))

for i in range(numberOfItems):
    list_of_weights.append(input("Enter the {}.items's weight : ".format(i+1)))
    list_of_values.append(input("Enter the {}.items's value : ".format(i+1)))

weights = tuple(list_of_weights)
values = tuple(list_of_values)

print('Weights :' , weights , 'Values :' , values)

def Converting(string):
    lists=[]
    lists[:0]=string
    return lists

def list_to_string(s):
    listToStr = ''.join(map(str, s))
    return listToStr
```

Figure 1.0

## 2) Actions

In “generate\_random\_state”, a state of 0 or 1s is generated up to the number of items. For example, if “numberOfItems” = 5, a state of 00110 is created and i kept the state values in the array named “listOfActions”. (For Hill Climbing)

```
class Knap_Sack(SearchProblem):  
  
    def generate_random_state(self):  
  
        state = ''  
        letter = '0','1'  
        for i in range(numberOfItems):  
            state = state + random.choice(letter)  
        return state  
  
    def actions(self, state):  
        listOfAction = []  
        for i in range(len(state)):   
            listOfAction.append(i+1)  
        a = list_to_string(listOfAction)  
        return a
```

Figure 1.1

## 3) Transition Model

In this section, i have determined how stats will take place in hill climbing. For example, if i have a state like 0011, the neighboring values can be 0111,1011,0010 or 0001. The reason for r-1 is because our actions start from 1, not 0. (For Hill Climbing)

```
def result(self, state, action):  
    State_list = Converting(state)  
    r = int(action)  
    if(State_list[r-1] == '0'):  
        State_list[r-1] = '1'  
        state = list_to_string(State_list)  
        return state  
    elif(State_list[r-1] == '1'):  
        State_list[r-1] = '0'  
        state = list_to_string(State_list)  
        return state
```

Figure 1.2

## 4) Objective Function

In the value part, if that item is exits, i look at its right and value. This process continues until i reach the capacity of the bag, then i return the total value. (For Hill Climbing)

```

def value(self, state):
    sum_w = 0
    sum_v = 0
    for i in range(numberOfItems):
        if(state[i] == 'I'):
            sum_w = sum_w + int(weights[i])
            sum_v = sum_v + int(values[i])
    if(sum_w <= knapsack_capacity):
        return sum_v
    else:
        return 0

```

Figure 1.3

## 5) Crossover and Mutation

In the genetic algorithm, the purpose of crossover is to obtain a new state with different sequences. In the code below, i randomly choose the index that i will do the cross process. The starting point, the part from the 0 index to the index i selected, shows state1, and the part from the index i have selected to the last index shows state2 then i combine these two states to get a new state.

In mutation, i convert the selected index value to 0 if it is 1, and to 1 if it is 0.

```

def crossover(self, state1, state2):
    part = random.randint(1, numberOfItems - 1)

    crossover_f_state1 = state1[:part]
    crossover_s_state2 = state2[part:]

    crossover_newstate = crossover_f_state1 + crossover_s_state2

    return crossover_newstate

def mutate(self, state):
    mut_number = random.randint(0, numberOfItems - 2)
    if(state[mut_number] == 0):
        state[mut_number] = 1
    elif(state[mut_number] == 1):
        state[mut_number] = 0
    return state

```

Figure 1.4

## 6) Making a Choice and Calling

```
95 print("Choose to algorithms.")
96 print("Enter 1 for Hill climbing")
97 #Requires: actions, result and value.
98 print("Enter 2 for Hill climbing Random Restarts")
99 #Requires: actions, result, value, and generate_random_state.
100 print("Enter 3 for Genetic")
101 #Requires: generate_random_state, crossover, mutate and value.
102 #-----
103 while True:
104     x = int(input("Which one? "))
105     if x == 1:
106         initial_type = int(input("\nEnter 1 to manually enter the initial state, 2 to randomize: "))
107         if initial_type == 1:
108             initial_state = input("Please enter (e.g. 10101)initial state: ")
109             while True:
110                 if (numberOfItems == len(initial_state)):
111                     initial_state_size = 1
112                     for x in range(numberOfItems):
113                         if initial_state[x] == '0' or initial_state[x] == '1':
114                             continue
115                         else:
116                             print("Wrong entry")
117                             initial_state_size = 0
118                             break
119                     if initial_state_size == 1:
120                         problem = Knap_Sack(initial_state)
121                         result = hill_climbing(problem, viewer = WebView())
122                         print(result.state)
123                         print(result.path())
124                         print('Stats : ')
125                         print(WebView().stats)
126                     elif initial_state_size == 0:
127                         print("No value other than zero and one is entered.")
128                         initial_state = input("Please enter (e.g. 10101)initial state: ")
129                         continue
130                 else:
131                     print("You entered wrong")
132                     print("The value you entered must be combined.")
133                     print("And its size has to be equal to the item number.")
134                     initial_state = input("Please enter (e.g. 10101)initial state: ")
135                     continue
```

Figure 1.5

3 options are offered to the user. These are Hill-Climbing, Hill-Climbing-Random-Restarts and Genetic. The first of these is Hill-Climbing. (Enter of 1) After that, two options are offered. The first is to manually enter the “initial\_state”. The other is the design of the system itself. If 2 is entered, the system generates and runs the code itself. If 1 is entered, the user is expected to correctly “initial\_state”. The correct format is that it's a string with size equal to the number of items and only 1's and 0's. (e.g. Item number is 5 so user should like enter 10101.)

```
133         elif initial_type == 2):
134             initial_state = ''
135             letter = '0', '1'
136             for i in range(numberOfItems):
137                 initial_state = initial_state + random.choice(letter)
138             problem = Knap_Sack(initial_state)
139             result = hill_climbing(problem, viewer = WebView())
140             print(result.state)
141             print(result.path())
142             print('Stats : ')
143             print(WebView().stats)
144         else:
145             print("You entered incorrectly ...")
146     #-----
147     elif x == 2:
148         problem = Knap_Sack(initial_state = None)
149         result = hill_climbing_random_restarts(problem, 10, viewer=WebView())
150         print(result.state)
151         print(result.path())
152         print('Stats : ')
153         print(WebView().stats)
154     #-----
155     elif x == 3:
156         problem = Knap_Sack(initial_state=None)
157         result = genetic(problem, population_size=100, mutation_chance=0.1, iterations_limit=0, viewer=WebView())
158         print(result.state)
159         print(result.path())
160         print('Stats : ')
161         print(WebView().stats)
162     #-----
163     if x == 1 or x == 2 or x == 3:
164         print("The End")
165         print("Developed by Ekin Şuataman, Mert Dumanlı and Ege Aydınoglu")
166         break
167     #-----
168     else:
169         print("Wrong value entered, please restart again...")
```

Figure 1.6

When the other two choices are made, the system tries to solve the problem with an empty “initial\_state”. I chose to make the hill-climbing-random-restarts to restart number 10. And, i choose to make genetic to population\_size 100, mutation\_chance 0.1 and iterations\_limit 0. It is sometimes difficult to fill in the iight and value parts. So, If the user enters an integer other than 1,2 and 3, i planned to give her/him rights again. In addition, i offered the opportunity to operate only once.

## C – Discussion on the Result

**Number of Items = 4**

**Bag Capacity = 12**

---

```
In [1]: runfile('D:/Masaüstü/Code.py', wdir='D:/Masaüstü')

How many items will you enter?4

What will be knapsack_capacity(Bag capacity)?12

Enter the 1.items's weight : 5

Enter the 1.items's value : 12

Enter the 2.items's weight : 3

Enter the 2.items's value : 5

Enter the 3.items's weight : 7

Enter the 3.items's value : 10

Enter the 4.items's weight : 2

Enter the 4.items's value : 7
Weights : ('5', '3', '7', '2') Values : ('12', '5', '10', '7')
Choose to algorithms.
Enter 1 for Hill climbing
Enter 2 for Hill climbing Random Restarts
Enter 3 for Genetic

Which one? |
```

Figure 2.0

I entered all values. And i entered 1 from keyboard. After that, i entered 1 again. I entered “0000” for “initial\_state”. And i ran the program with the ib browser.



```
Choose to algorithms.  
Enter 1 for Hill climbing  
Enter 2 for Hill climbing Random Restarts  
Enter 3 for Genetic  
  
Which one? 1  
  
Enter 1 to manually enter the initial state, 2 to randomize: 1  
  
Please enter (e.g. 10101)initial state: 0000  
Starting the WebViewer, access it from your web browser, navigating to the  
address:  
http://localhost:8000  
To stop the WebViewer, use the "Stop running" link (on the viewer site, from the  
browser)
```

Figure 2.1



Figure 2.2 (initial\_state = 0000)

And the program found us this result. This result means that buying the first and third item is the most profitable result. But there is a problem here. After all, it doesn't find the right answer for us. Because it gets stuck at local max value and cannot reach the real global max value. Based on this, hill climbing may not always return the optimal result. Random-restart hill climbing can be used to solve this. Eventually, I can say that the starting point is very important in the hill-climbing search algorithm. After that, I ran other algorithms. I ran it with random numbers and found the global max with the hill-climbing search.



Figure 2.3 (initial\_state was random)

I reached the same results, with the same inputs, in the other two algorithms.

The process of reaching the solution is given in Figure 2.4 with Hill-Climbing-Random-Restarts.

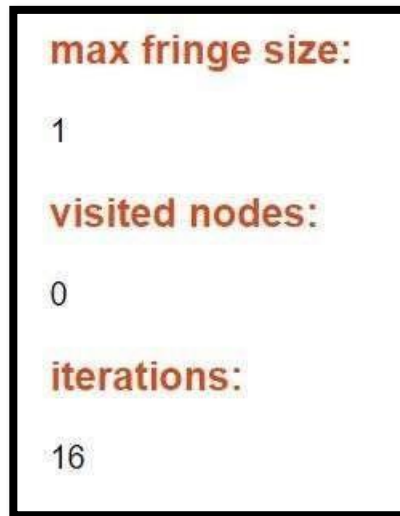


Figure 2.4

The process of reaching the solution is given in Figure 2.5 with Genetic.

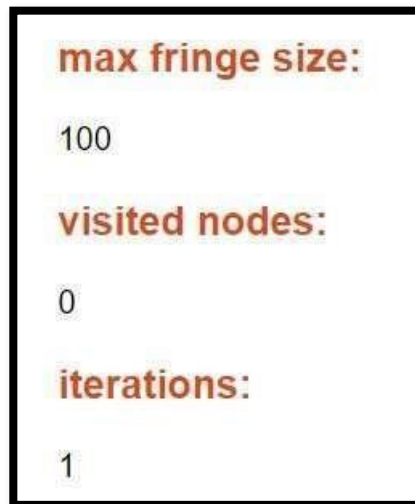


Figure 2.5



**Number of Items = 15**

**Bag Capacity = 50**

```
Enter the 15.items's weight : 4

Enter the 15.items's value : 60
Weights : ('24', '10', '10', '7', '2', '8', '6', '5', '9', '12', '20', '18',
'13', '5', '4') Values : ('50', '10', '25', '30', '20', '25', '40', '15', '12',
'22', '35', '45', '55', '100', '60')
Choose to algorithms.
Enter 1 for Hill climbing
Enter 2 for Hill climbing Random Restarts
Enter 3 for Genetic

Which one? 1

Enter 1 to manually enter the initial state, 2 to randomize: 1

Please enter (e.g. 10101)initial state: 000000000011111
Starting the WebViewer, access it from your web browser, navigating to the
address:
http://localhost:8000
```

Figure 3.0

I entered all values. And i entered 1 from keyboard. After that, i entered 1 again. I entered “000000000011111” for “initial\_state”. And i ran the program with the ib browser.

**started:**

Algorithm just started.

**new\_iteration:**

New iteration with 1 elements in the fringe: [Node <000000000011111>]

**expanded:**

Expanded [Node <000000000011111>] Successors: [[Node <100000000011111>, Node <010000000011111>, Node <001000000011111>, Node <000100000011111>, Node <000010000011111>, Node <000001000011111>, Node <000000100011111>, Node <000000010011111>, Node <000000001011111>, Node <100000000011111>, Node <0000000000111110>, Node <100000000011111>, Node <1000000000011111>, Node <1000000000011111>, Node <010000000011111>, Node <100000000011111>, Node <001000000011111>, Node <100000000011111>, Node <000010000011111>]]

**finished:**

Finished algorithm returning Node <000010000011111>. Solution type: returned after not being able to improve solution

Figure 3.1

And this time i fell into the local max trap and this time i could not find a close result.

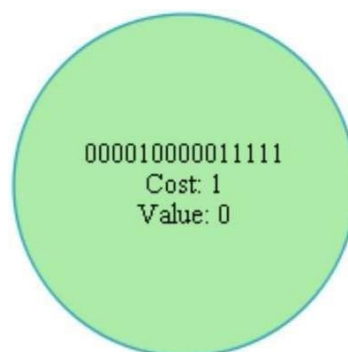


Figure 3.2 (Result of Figure 3.0 process)

Again, i ran it with generic and this time generic did not give us the real result. But it gave a better result than hill-climbing.

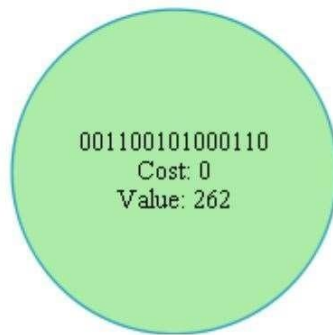


Figure 3.3 (Genetic result)

There is more randomness in the genetic algorithm than hill climbing, which shows us that the genetic algorithm is more advantageous. But despite these, genetic algorithm is still not complete and optimal either.

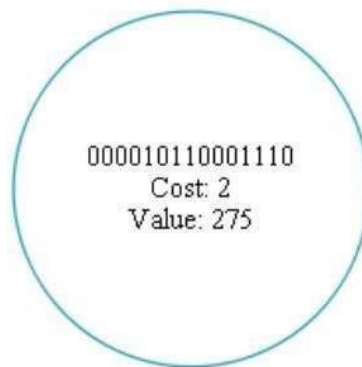


Figure 3.4 (Hill\_Climbing\_Random\_Restarts result)

Another time i used the search for “Hill\_Climbing\_Random\_Restarts” for the same values. I achieved slightly better results. But this time, our restarts\_limit number prevented us. Also hill climbing and hill climbing random restart spends more time than the genetic algorithm. That's why their time values and time complexities are higher.

I could not reach the real value in our 3 trials. I found the actual value from the code i got from the site you referenced. \*\*\*

```

1  def knapSack(W, wt, val, n):
2      if n == 0 or W == 0:
3          return 0
4      if (wt[n-1] > W):
5          return knapSack(W, wt, val, n-1)
6      else:
7          return max(
8              val[n-1] + knapSack(
9                  W-wt[n-1], wt, val, n-1),
10             knapSack(W, wt, val, n-1))
11  wt = [24, 10, 10, 7, 2, 8, 6, 5, 9, 12, 20, 18, 13, 5, 4]
12  val = [50, 10, 25, 30, 20, 25, 40, 15, 12, 22, 35, 45, 55, 100, 60]
13
14  W = 50
15  n = len(val)
16
17  a = knapSack(W, wt, val, n)
18  print(a)

```





Figure 3.5

True Result is 345 according to this code.

## Notes

- ☐ According to us, the most difficult part in doing this homework was the problem definition.
- ☐ I used the Spyder 4.1.4 version. (Python 3.8.3 64-bit | Qt 5.9.7 | PyQt 5.9.2 | Windows 10)
- ☐ When running and trying the code, if the code that is run once is processed with a ib browser, it cannot be processed again. I want to say that, if the operation was performed on <http://localhost:8000> site, i had to close both the “Spyder 3.8” and the ib browser in order to make a new operation from scratch. Otherwise, the ib browser just shows us the result of the most recent operation repeatedly. This made it very difficult for us to test the code. At this point i understood the importance of processing pair.

## References

-  <https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>
-  **Stuart Russell and Peter Norvig;**  
**Artificial Intelligence: A Modern Approach, Pearson, 2017**
-  <https://www.jquery-az.com/3-ways-convert-python-list-string-join-map-str/>
-  <https://stackoverflow.com/questions/21517024/how-to-join-values-of-map-in-python>